Experimental Evaluation of Deep Reinforcement Learning Algorithms

Nikola Mrzljak, Tomislav Hrkać
Department of Electronics, Microelectronics, Computer and Intelligent Systems
University of Zagreb Faculty of Electrical Engineering and Computing
Zagreb, Croatia
nikolamrzljak@yahoo.com, tomislav.hrkac@fer.hr

***Abstract*—Reinforcement learning is an interesting field with regards to its applicability. As such, it has been studied for use in computer vision where it has just recently found its place. The problem it encountered was the high dimensionality of input data in computer vision. The aim of this paper is to provide an overview of the reinforcement learning algorithms for solving the task of reinforcement learning with raw pixels of an image as an input to the algorithm and test their performance on Atari Breakout video game. The comparison of different algorithms will be given and discussed.**

I. Introduction

Reinforcement learning is an important branch of machine learning with many potential applications, such as robot navigation and control, autonomous driving, operations research, human-computer interaction, automated game playing and many more [1]–[3].

In reinforcement learning setting, an agent is placed in an environment and has to take a sequence of actions in order to achieve a goal and/or to avoid undesired results. More generally, for each transition from one state to another by a given action, a numerical reward is specified (which can also be negative if a certain state is undesired, or zero - in this case, it is usual to say that there is no reward) and the task of the agent is to maximize the sum of the received rewards by taking appropriate actions. In each step, the agent observes the state of the environment and outputs a decision which action to take.

In recent years, deep learning has enabled an unprecedented progress in many areas of machine learning, especially in computer vision, enabling significant progress in image classification [5]–[8], object detection [9]–[11], semantic segmentation [12]–[14], etc. More recently, starting with DeepMind [15], this success has been carried over to reinforcement learning, resulting in several successful deep reinforcement learning (DRL) algorithms that can learn to map raw input images to actions, thus connecting reinforcement learning with computer vision. Good overviews of recent deep reinforcement learning algorithms from a theoretical point of view can be found in [16], [17].

In contrast with the above survey papers, the focus of this paper is an experimental evaluation of several state-of-the-art deep reinforcement learning algorithms, as well as exploration strategies and techniques for storing and selecting past experiences. A common testbed for performance evaluation of deep reinforcement learning algorithms is automatic video-game playing; therefore, for our experimental evaluation, we use one of the most commonly used games for such purposes, namely the Atari Breakout game. The inputs to the tested algorithms are only the frames of the game screen and the information about the current game score. As a metrics for comparing the performance of the algorithms we use the achieved game score and the episode length as functions of the number of episodes played during the learning. Compared to the other real world problems, the Atari Breakout game is really well structured with no random noise affecting the input space (game screen). E.g. in autonomous car driving partial blockage of the image can be present, greatly affecting the performance and significantly increasing the problem complexity.

In section II, after a brief overview of general deep learning principles, we briefly describe the evaluated algorithms and their variations. In section III, the experimental setup and results of the testing are presented. Conclusions and possible directions for future work are given in section IV.

II. Deep reinforcement learning

A. Background

The task of reinforcement learning can be modeled by means of Markov decision process (MDP), formally defined as a tuple:

$$MDP = (S, A, P(s_{t+1}|s_t, a_t), R(s_{t+1}|s_t, a_t), \gamma),$$

where S is a set of states, A is a set of actions, $P(s_{t+1}|s_t, a_t)$ models the probability of transition to future state s_{t+1} from current state s_t, $R(s_{t+1}|s_t, a_t)$ (also commonly denoted by r_{t+1}) are the rewards for the given transition and γ is a discount factor for a given environment. The discount factor γ models the delayed reward mechanism that enables an agent to give more weight to the future rewards instead to the rewards closer to the present.

In MDP, at time step t, the agent is in the state s_t, chooses to do action a_t and, as a consequence of this action, it transitions to the state s_{t+1} with probability $P(s_{t+1}|s_t, a_t)$ for which it is rewarded with reward $r_{t+1} = R(s_{t+1}|s_t, a_t)$.

The problem at hand is to maximize the cumulative discounted reward for a given state s defined as:

$$\mathbf{R}(s) = \sum_{t=1}^{n} \gamma^t r_{t+1}.$$

https://doi.org/10.20532/ccvw.2018.0003
CCVW 2018, Oral Session

In the previous equation when t is equal to n the process is terminated (since we are interested in the finite process). To solve this problem, various algorithms are available, which can be classified in two main categories: value-based (e.g. Q-learning) and policy-based (e.g. policy gradient). A policy is a sequence of actions an agent takes when transitioning from one state to another. Policy can be implicit or explicit. Explicit policy can be represented by a probability distribution, in which case for a given state s_t the probability of choosing action a is given, which constitutes policy $\pi(a|s_t)$. Implicit policy can be defined as choosing the action a that gives the highest value of Q function.

Action-value function $Q^\pi(s_t, a_t)$ is a function that models how well it is to be in a state s_t and perform action a_t in it and afterwards behave according to policy π.

Q-learning is an algorithm that is based on approximating $Q(s_t, a_t)$. $Q(s_t, a_t)$ can be easily calculated by using the well-known Bellman equation that can be applied to the optimal action-value function $Q^*(s_t, a_t) = \max_\pi Q^\pi(s_t, a_t)$.

Bellman equation has been shown to converge to the optimal value $Q^*(s_t, a_t)$ as the number of iterations approaches the infinity [4]. The Q function can be parametrized by a model with parameters θ. This can be denoted as $Q(s_t, a_t|\theta)$. In this case, target value y of $Q(s_t, a_t|\theta)$ for a given pair (s_t, a_t) is defined as:

$$y = \begin{cases} r_{t+1}, & \text{if } s_{t+1} \text{ is terminal} \\ r_{t+1} + \gamma \max\limits_{a_{t+1}} Q(s_{t+1}, a_{t+1}|\theta) & \text{otherwise} \end{cases}$$

With respect to this, the loss for a given experience $(s_t, a_t, r_{t+1}, s_{t+1})$ is defined as:

$$L = \frac{1}{2}(y - Q(s_t, a_t|\theta))^2.$$

Parameters of a model defined like this can be updated by the following rule:

$$\theta_{t+1} = \theta_t - \alpha(y_t - Q(s_t, a_t|\theta_t)),$$

where α represents the learning rate. In this case, the expectation has been approximated by its estimate.

Q-learning algorithm is a greedy algorithm with respect to choosing actions that it will perform since the policy is greedy. In practice, ϵ-greedy state exploration approach is used in order to enable better state exploration. ϵ-greedy strategy selects random action with probability ϵ and the best one with probability 1-ϵ. Another state exploration strategy is Boltzmann's strategy, which, being probabilistic, provides better state exploration since it takes into consideration every action in every time step. The probability of taking action a in state s is, therefore:

$$p(a) = \frac{\exp(Q(s, a))}{\sum_i \exp(Q(s, a_i))}.$$

Policy gradient is a method that optimizes policy $\pi(a|s, \theta)$. To perform the policy optimization, some optimization function must be defined. In this case, the optimization function is the expectation of the delayed reward which is done over all of the trajectories τ. Trajectory τ is defined as $(s_0, a_0, r_1, s_1, ..., s_{T-1}, a_{T-1}, r_T, s_T)$. Formal definition of the function that is subject to optimization is given by:

$$\mathbb{E}_\tau[R_\tau|\pi, \theta] = \int_\tau P(\tau|\pi, \theta) R_\tau d\tau.$$

R_τ is the sum of rewards gained throughout the trajectory τ. Since the goal is to get the greatest possible amount of reward, to optimize the goal function means to maximize the given function. In order to maximize the function, gradient must be calculated. The value of the gradient is:

$$\nabla_\theta \mathbb{E}_\tau[R_\tau|\pi, \theta] = \mathbb{E}_\tau[R_\tau \nabla_\theta \log P(\tau|\pi, \theta)].$$

From that follows that the estimator of the gradient g is: $\hat{g}(\tau) = R_\tau \nabla_\theta \log P(\tau|\pi, \theta)$ Estimator can be used for estimating $\nabla_\theta \mathbb{E}_\tau[R_\tau|\pi, \theta]$. Since $P(\tau|\pi, \theta)$ depends on the model of the environment, $\nabla_\theta \mathbb{E}_\tau[R_\tau|\pi, \theta]$ can't be used for calculating the gradient. Before using the aforementioned estimator, $P(\tau|\pi, \theta)$ has to be parsed with respect to the definition of MDP. After the parsing, and noting that $P(a_i|s_i, \pi, \theta) = \pi(a_i|s_i, \theta)$, the resulting estimator is:

$$\hat{g}(\tau) = R_\tau \nabla_\theta \log P(\tau|\pi, \theta) = R_\tau \nabla_\theta \sum_{t=0}^{T-1} \log \pi(a_t|s_t, \theta). \tag{1}$$

With the given model definition, parameters are updated by using the following expression:

$$\theta = \theta + \alpha \cdot \hat{g}(\tau).$$

B. Deep reinforcement learning algorithms

The goal of deep reinforcement learning is to learn a model with parameters θ to behave optimally in an environment where the only input is an image. A number of algorithms have been designed to solve this problem. In this paper we consider the following algorithms: Deep Q Network (section II-B1), Double Deep Q Network (section II-B2), Dueling Double Deep Q Network (section II-B3) and Asynchronous advantage actor-critic (section II-B4). The first three algorithms are value-based, while the last one is policy-based.

In most of the deep learning tasks the data is independent, whereas in the reinforcement learning the problem of highly correlated samples is present. In order to solve this problem, experience replay mechanism is introduced as stated in [15]. In this paper, the following modification of this approach was used: whenever the experience is going to be removed from the buffer, it is checked whether it has a non-zero reward. If it has positive reward it is again added to the buffer with a probability α whereas if it has a negative reward it is added to the buffer with a probability β. The reason for introducing this modification is that when the inspection of the experiences presented to the agent was done it has been noticed that the number of rewarded experiences was very low. This mechanism, called prioritized forgetting, was introduced to prevent the removal of such, already rare experiences, and it showed very good results.

1) Deep Q Network (DQN): Deep Q network is the simplest successful algorithm based on Q-learning. This algorithm has been used with ϵ-greedy exploration strategy in the original work but can be easily combined with other exploration strategies such as Boltzmann's. Detailed description and pseudocode for this algorithm can be found in [15].

2) Double Deep Q Network (DDQN): This algorithm is an upgrade to the aforementioned DQN algorithm.

It has been shown that DQN is overoptimistic in its approximation of the value of Q function. This problem is present in every model that estimates the value of the best option, as shown in [18]. This is exactly what DQN algorithm does according to equation $Q(s_t, a_t) = r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$. Over-optimism has also been shown to be a good exploration strategy when facing unknown situations. Over-optimism wouldn't present a problem if the estimates were uniformly distributed over all of the states. However, if overoptimistic estimates are concentrated around poor states, it can lead to suboptimal policies.

This algorithm has empirically shown that DQN's poor policies are due to the fact it overestimates the Q function.

One of the possible reasons for overestimating the value of Q function in the DQN algorithm is using the same model for both estimating Q function value of the following state and selecting an action [19]. The way the DDQN deals with the aforementioned problem is by introducing another, target model, with parameters θ', that is used for estimating the value of Q function while the primary model with parameters θ is used for action selection. In this way, function estimation is decoupled from action selection and this eliminates coupling as a reason of overestimation. Function approximation is done by using the following equation:

$$Q(s_t, a_t|\theta) = R(s_{t+1}|s_t, a_t) + \gamma Q(s_{t+1}, \mathrm{argmax}_{a_{t+1}} Q(s_{t+1}, a_{t+1}|\theta)|\theta^{'}).$$

The target network needs to be updated too. It can be updated using some of the following strategies: periodic model swapping, periodic update of target network to the values of the primary network and linear interpolation of parameters of target model towards the parameters of the primary network ($\theta^{'} = \theta^{'} + \alpha(\theta - \theta^{'})$). There is no optimal choice with respect to how to update the target network. However, it has been shown by [20] that periodic update improves algorithm stability.

According to [21], it has been empirically proved that double Q-learning is unbiased while basic Q-learning is biased. It has also been shown that correcting the overestimation error resulted in better policies.

3) Dueling Double Deep Q Network (Dueling DDQN): Dueling Double Deep Q Network is an upgrade to the above described Double Deep Q Network. This algorithm doesn't bring any novelties with respect to the algorithm but rather changes the representation of the Q function. The Q function can be written as a combination of two functions: the advantage function and the value function. The advantage function is defined as:

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \quad (2)$$

Simply put, the value function measures how good is it to be in a given state s_t, while the advantage function measures how good it is to perform action a_t while in state s_t. The advantage function can, therefore, prioritize some action over other actions in a given state.

From equation (2), it follows that Q function can be written as:

$$Q^{\pi}(s_t, a_t) = V^{\pi}(s_t) + A^{\pi}(s_t, a_t) \quad (3)$$

Dueling DDQN builds its architecture using (3). Input convolutional layers are the same as in DQN and DDQN while changes are made in the fully connected layers. The fundamental change is that the output of the last convolutional layer is shared between two streams, one for modeling value function $V(s|\theta, \alpha)$ and another one for modeling advantage function $A(s, a|\theta, \beta)$. α and β represent parameters of fully connected layers for value function and advantage function, respectively. θ represents parameters of convolutional layers that are shared by both functions.

The problem with implementing equation (3) straightforward as it is is that, given the value of $Q^{\pi}(s_t, a_t)$, the values of $V^{\pi}(s_t)$ and $A^{\pi}(s_t, a_t)$ can't be determined uniquely. In order to deal with this problem, the advantage function is replaced with a substitution function A_z so that the final equation is:

$$Q(s_t, a_t) = V(s_t) + A_z(s_t, a_t)$$

The function A_z can be defined in many ways but the following one proved very good in practice:

$$A_z(s_t, a_t) = A(s_t, a_t) - \frac{1}{m} \sum_{a_t^{'}} A(s_t, a_t^{'}) \quad (4)$$

In equation (4), m is the number of actions in the given environment. Introduction of the substitution function A_z results in a loss of the original semantics of value and advantage functions but it gives greater algorithm stability, as shown in [22].

4) Asynchronous advantage actor-critic (A3C): Asynchronous advantage actor-critic algorithm [26] is the only policy iteration algorithm in this paper. The main reason policy-based algorithms weren't previously used in deep reinforcement learning is that they demand neighboring frames that are highly correlated, caused by their temporal dependency.

The fundamental idea used in the A3C algorithm to solve this problem is to start multiple agents with the same parameters in different environments and let them interact with the environment. In this way, the correlation of the updates is decreased.

Another difference with respect to previous algorithms is that this algorithm speeds up the propagation of reward by using n-step returns [23] whereas previous algorithms used only one time step for updating parameters.

The number of steps before updating the algorithm parameters is determined by t_{max} or by terminal state. Expression (1) is an unbiased estimator but there is a way to reduce it's variance as stated in [24]. The final estimate of the gradient of the goal function for the A3C algorithm is:

$$\hat{g}(\tau) = \nabla_\theta \sum_{t=0}^{t_{max}-1} \log \pi(a_t|s_t, \theta_\pi) A(s_t, a_t|\theta_v)$$

θ_π denotes the parameters of the policy while θ_v denotes the parameters of the value function. In practice, all layers are shared except for the last fully connected layers which are separate since one of them models the value function with linear function as the activation, while the other one models the policy with softmax function as the activation.

It has been shown by [25] that it is very useful to introduce entropy into the goal function since it prevents early convergence of the policy into some suboptimal policy. In order to regulate the influence of entropy function on the optimization process, hyperparameter β is introduced to scale its influence. If the influence of the entropy is too big the resulting policy becomes stochastic.

Detailed description and pseudo-code for this algorithm can be found in [26].

III. Experimental results

Algorithm learning and evaluation was done using OpenAI Gym's [27] implementation of Atari Breakout game where it was noted that the average human score equals to 31, as stated by [15]. The evaluation was restricted only to this game due to limited available resources and the length of the training process; however, previous work has shown that the relative performance of the tested algorithms is similar for most of the games from the Atari 2600 collection [15], [21], [22], [26]. Since the environment gives the state represented as an image of height 210 pixels and width of 160 pixels in RGB format, preprocessing was done. The preprocessing consists of resizing the image height and width to 84 pixels and converting it to grayscale format. In all of the algorithms, concatenation of the 3 latest frames was used to represent the current state. In the case of the start of the game, the first frame is repeated 3 times. The only algorithm that used only the current frame as input is A3C with LSTM layer. Value-based algorithms used prioritized forgetting for memory replay. Experiments were done using Nvidia GTX 1060 6GB and Nvidia GTX 1080 graphics cards. Every point in the graph (for both reward and episode length) represents the average of five consecutive games being played. In the terms of evaluation, the episode stands for one game played, the episode length is the number of time steps in the episode (number of interactions with the environment) and the reward is the sum of the rewards received in each time step (cumulative reward). In the table I the used architectures are displayed.

TABLE I
Table of architectures

Layer	DQN	Dueling DDQN	A3C	LSTM A3C
1	Input 84x84	As DQN	As DQN	As DQN
2	Conv 32 8x8 4x4 stride ReLU activation	As DQN	As DQN	As DQN
3	Conv 64 4x4 2x2 stride ReLU activation	As DQN	As DQN	As DQN
4	Conv 64 3x3 1x1 stride ReLU activation	As DQN	As DQN	As DQN
5	Flatten	As DQN	As DQN	As DQN
6	FC (512)	Advantage stream (FC($\|A\|$)) Value stream FC(1)	FC (512)	As DQN
7	Output FC($\|A\|$)	Output (Sum layer)	Policy stream (FC($\|A\|$) Value stream (FC(1)	Recurrent layer (256)
8	-	-	Output (Policy stream)	As 7th layer of A3C
9	-	-	-	Output (Policy stream)

A. Deep Q Network

DQN was modeled using a convolutional neural network. Input image dimensions were 84x84. The network is made of a convolutional layer with 32 kernels of size 8 in both dimensions, a stride of 4 in both dimensions without padding and ReLU activation. This is the input layer. The following layer is also convolutional with 64 kernels of size 4 with a stride of 2 without padding and ReLU activation. Next layer is convolutional with 64 kernels of size 3, a stride of 1 without padding and ReLu activation. These layers are followed by a flattening layer and two fully connected layers. The first fully connected layer has 512 neurons and ReLU activation. The last layer has as many outputs as there are game actions and has a linear activation function since it models $Q(s, a)$.

The experiment was done using ϵ-greedy state exploration strategy. The training was done for 4 million iterations and it lasted 90 hours using Nvidia GTX 1080 graphics card. The results are displayed in Fig. 1.

As it can be seen from the graph of the episode length and reward, the agent is slowly learning how to play the game.

B. Double Deep Q Network

The architecture that was used is the same as for DQN.

The experiment was done using ϵ-greedy state exploration strategy. The training lasted 17 hours using Nvidia GTX 1080 graphics card. The results that were obtained are given in the Fig. 2.

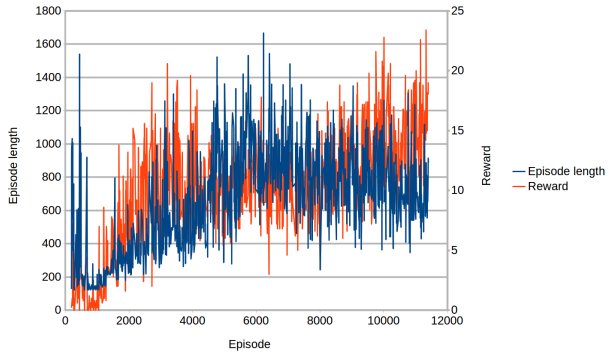

Fig. 1. DQN: Display of episode length and reward with respect to the episode count

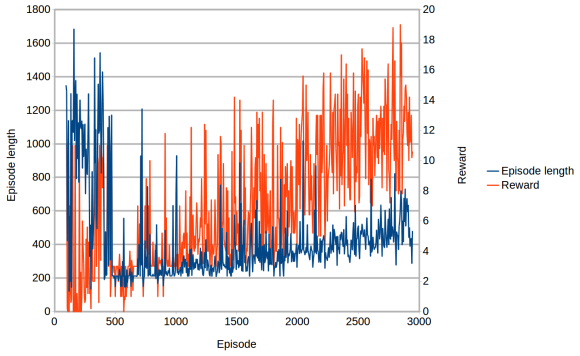

Fig. 2. DDQN: Display of episode length and reward with respect to the episode count

From the display of both the episode length and the reward value, it can be deduced that the agent is learning how to play the game since both are increasing.

C. Dueling Double Deep Q Network

The only difference between the architecture of this network and the previous ones is that the fully connected layer with 512 neurons is removed and, based on the output of the flattening layer, two separate streams are created: one for approximating the value and one for approximating the advantage function.

Two approaches to state exploration were used in combination with this network, ϵ-greedy and Boltzmann.

The results that were obtained by using Boltzmann exploration strategy are shown in Fig. 3. The training lasted 24 hours using Nvidia GTX 1080 graphics card.

The results that were obtained by using ϵ-greedy exploration strategy are given in Fig. 4. The training was done for 2 million iterations and it lasted 61 hours using Nvidia GTX 1060 6GB graphics card.

Comparing the results of the Boltzmann version of Dueling DDQN with the ϵ-greedy one shows that ϵ-greedy one proved better because it resulted in greater reward value. However, when testing, Boltzmann approach showed to be more consistent (stable) with respect to how big reward it was getting from the environment while the ϵ-greedy approach had greater variance.

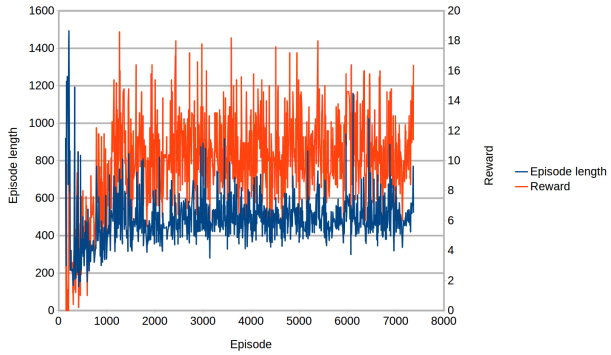

Fig. 3. Boltzmann Dueling DDQN: Display of episode length and reward with respect to the episode count

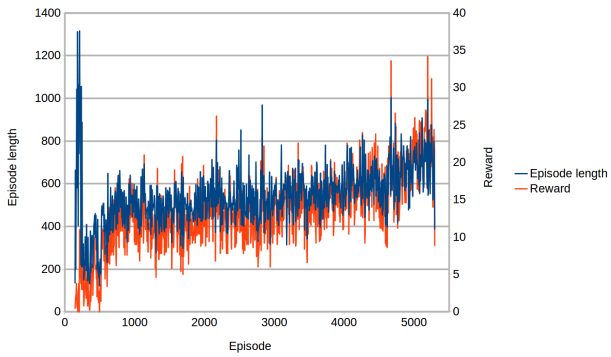

Fig. 4. ϵ-greedy Dueling DDQN: Display of episode length and reward with respect to the episode count

D. Asynchronous advantage actor-critic

This algorithm has been tested with two different architectures. Both architectures were similar to the one that was used in DQN. The only difference is that after the first fully connected layer, either LSTM or additional fully connected layers (with one neuron with linear activation for value approximation and a number of neurons equal to the number of actions with softmax activations for policy approximation) are introduced. The outputs of these additional layers are used as a stream for value and policy. Both algorithms used 24 threads as agents. It must be remarked that the graphs are displayed for only one thread/agent. In other words, this algorithm roughly played 24 times more games than it is displayed on the graph.

The results that were obtained by using the LSTM layer are given in Fig. 5. The training lasted 32 hours using Nvidia GTX 1060 6GB graphics card.

The results that were obtained by a simple feedforward convolutional network are shown in Fig. 6. The training lasted 24 hours using Nvidia GTX 1060 6GB graphics card.

From the display of both the episode length and the reward value of both LSTM A3C algorithm and feedforward A3C algorithm, it can be deduced that the agent is learning how to play the game since both are increasing. It can also be stated that in this case the feedforward A3C algorithm proved to learn better policy.

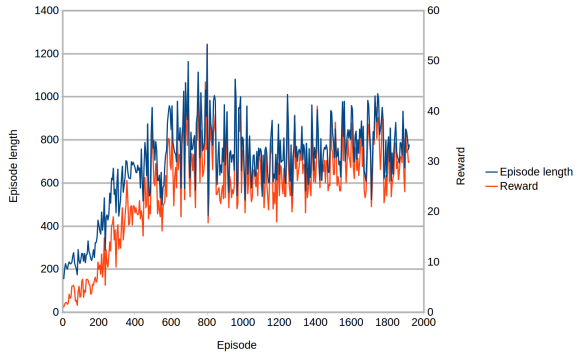

Fig. 5. LSTM A3C: Display of episode length and reward with respect to the episode count

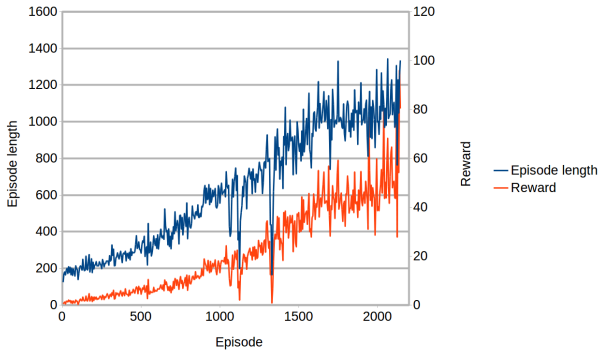

Fig. 6. Feedforward A3C: Display of episode length and reward with respect to the episode count

IV. Conclusion

The main focus of this paper was to provide an overview of various deep reinforcement learning techniques. By comparing the acquired results it can be seen that the A3C algorithm has proved to be the best solution so far since it has yielded the greatest reward in the shortest time span. With this in mind, it can be noted that this could be the algorithm that will be used as the cornerstone of future work.

References

[1] G. Barto, P. S. Thomas, and R. S. Sutton, “Some Recent Applications of Reinforcement Learning”. In Proceedings of the Eighteenth Yale Workshop on Adaptive and Learning Systems, 2017.

[2] J. Kober, J. Andrew (Drew) Bagnell and J. Peters, “Reinforcement Learning in Robotics: A Survey”. International Journal of Robotics Research, July 2013.

[3] A. El Sallab, M. Abdou, E. Perot and S. Yogamani “Deep Reinforcement Learning for Autonomous Driving. Electronic Imaging, Autonomous Vehicles and Machines 2017, pp. 70-76(7).

[4] R. Sutton and A. Barto, “Reinforcement Learning: An Introduction,” MIT Press, 1998.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks”. Advances in Neural Information Processing Systems 25, pp 1097–1105., 2012.

[6] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition”. CoRR, abs/1409.1556, 2014.

[7] K. He, X. Zhang, S. Ren and J. Sun, “Deep Residual Learning for Image Recognition”. 10.1109/CVPR.2016.90, 2015.

[8] C. Szegedy, W. Liu, Y Jia, P. Sermanet and S. Reed, “GoogLeNet: Going Deeper with Convolutions”. CVPR 2015., pp. 1-9

[9] S. Ren, K. He, R. Girshick and J. Sun, “Faster R-CNN: towards real-time object detection with region proposal networks”. NIPS 2015, pp. 91-99.

[10] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection”. CVPR 2016., pp. 779-788.

[11] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C-Y Fu, A. C. Berg, “SSD: Single Shot MultiBox Detector”. ECCV 2016.

[12] J. Long, E. Shelhamer and T. Darrell, “Fully convolutional networks for semantic segmentation”. In Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp 3431–3440, 2015.

[13] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation”, pp 234–241. Springer International Publishing, Cham, 2015.

[14] V. Badrinarayanan, A. Kendall and R. Cipolla, “SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation”. IEEE TPAMI, vol. 39, Issue 12, 2017., pp. 2481-2495.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning,” arXiv, 2013

[16] Y. Li, “Deep Reinforcement Learning: An Overview”. ArXiv, 2017.

[17] K. Arulkumaran, M. P. Deisenroth, M. Brundage and A. A. Barath, “A Brief Survey of Deep Reinforcement Learning”, IEEE Signal Processing Magazine, 2017

[18] J.E. Smith and R.L. Winkler, “The Optimizer’s Curse: Skepticism and Postdecision Surprise in Decision Analysis”, 2006.

[19] H. van Hasselt, “Double Q-learning”, 2010.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis, “Human-level control through deep reinforcement learning,” 10.1038/nature14236, 2015

[21] H. van Hasselt, A. Guez and D. Silver, “Deep Reinforcement Learning with Double Q-learning,” arXiv, 2015.

[22] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot and N. de Freitas,“Dueling Network Architectures for Deep Reinforcement Learning,” arXiv, 2016.

[23] J. Peng and R.J. Williams, “Incremental multi-step q-learning,” Machine Learning, 1996.

[24] R.J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning”, 1992.

[25] R.J. Williams and J. Peng, “Function Optimization Using Connectionist Reinforcement Learning Algorithms”, 1991.

[26] V. Mnih, A.P. Badia, A. Graves, T.P. Lillicrap, T. Harley, D. Silver and K. Kavukcuoglu, “Asynchronous Methods for Deep Reinforcement Learning”, arXiv, 2016.

[27] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba “OpenAI Gym”, arXiv:1606.01540, 2016.

[28] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, M. Bowling “Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents”, doi:10.1613/jair.5699, 2013.